Inlupp 3: Avancerade datastrukturer

Imperativ Programmering DT501G, HT 2019
4 december 2019

Deadline

Deadline för inlämning är **13 december**.

Muntlig redovisning

Förutom att lämna in din källkod ska du redovisa din inlämning muntligt för din handledare (Daniel respektive Quantao). När den skriftliga rapporten (med källkod) har rättats kommer labbhandledaren att skicka en doodle-länk där du kan boka tid för den muntliga redovisningen. Du ska då muntligt kunna svara på några frågor om dina lösningar.

Lärandemål

Målet med uppgifterna i den här inlämningsuppgiften är att

- förstå användningsområden för länkade listor, samt att kunna implementera och använda dem,
- behärska minnesallokering (`malloc` och `free`),
- behärska rekursion,
- hantera abstrakta datatyper.

Testfall

Precis som i inlupp 2 måste du redovisa minst två testfall för varje uppgift, som du har använt för att kontrollera att programmet gör som det ska. Testfallen ska testa olika saker (inte bara två varianter av giltigt input, t ex). Ange dina testfall, och förväntat resultat, som kommentarer i din källkod.

Dokumentation

Glöm inte heller att dokumentera dina funktioner genom att beskriva vad de gör, vad de förväntar sig för input, och vad de ger för output i olika fall.

Uppgifter

3.1 Länkad lista

Tänk dig att du ska implementera ett äventyrsspel, med en spelare som kan gå runt i världen och plocka på sig olika saker. Då behöver du också implementera en lista med de saker som spelaren bär med sig. Här ska du implementera en länkad lista för det syftet.

För att underlätta problemet kan vi tänka oss att alla "saker" i spelet har ett visst ID-nummer. I praktiken skulle man kanske vilja implementera en **`struct`** som kan representera flera olika egenskaper hos en "sak" (namn, vikt, etc), men här nöjer vi oss med heltal: sak nr 1, sak nr 2, och så vidare.

Implementera datastrukturen

Implementera en länkad lista som innehåller heltal. För att göra detta behöver du en **`struct`** som implementerar ett element i listan (med dess värde, och pekare till nästa element). Gör också en **`typedef`** för din **`struct`**.

Bygg listan framifrån

Implementera också en funktion som lägger till ett listelement före ett annat, enligt kodlistning 1. Glöm inte att kontrollera att det gick att allokera ett nytt listelement i din funktion (det vill säga att anropet till `malloc` lyckades). Titta gärna tillbaka på exemplen från föreläsning 6 när du skriver din kod, om du känner dig vilsen.

Efter att ha implementerat detta borde du kunna implementera följande kodsnutt (i `main`, t ex).

```
/*
 * Lägger till ett tal först i en lista.
 *
 * Parametrar:
 *
 * list
 * Pekare på det element innan vilket det nya ska läggas in
 * (alltså listans huvud). Det kan vara en tom lista.
 *
 * data
 * Det nya värdet.
 *
 * Returnerar:
 * En pekare till listans huvud, om det lyckades.
 * NULL annars.
 */
int_list * push_front( int_list * list, int data);
```

Kodlistning 1: Specifikation för `push_front`.

```
int_list * head = NULL;        // Skapa tom lista: bara en NULL-pekare.
head = push_front( head, 10 ); // Nu är listan [10].
head = push_front( head, 20 ); // Nu är listan [20, 10].
head = push_front( head, 30 ); // Nu är listan [30, 20, 10].
```

Enligt specen i kodlistning 1 tar alltså funktionen en `int_list*` som argument och returnerar också en `int_list*`. Det kanske verkar konstigt vid första anblicken, men anledningen är att eftersom listan kommer att få ett nytt första element måste vi alltså ändra på pekaren `list` som skickas som in-argument till funktionen. På samma sätt som att man inte kan ändra på en **`int`** som skickas till en funktion utan att skicka en pekare **`int`**`*` till den i stället, så kan vi heller inte ändra en pekare `int_list*` utan att skicka en pekare till pekaren, alltså en `int_list**`! Då är det enklare att *returnera* den nya pekaren i stället.

Traversera listan med rekursion

För att göra något med din listimplementation behöver du också ha ett sätt att gå igenom listan från början till slut och göra något med varje element. För länkade listor lämpar sig rekursiva funktioner bra för att göra det.

Implementera en funktion `print_list` som skriver ut innehållet i en lista som du har konstruerat med koden ovan. Eftersom en länkad lista inte nödvändigtvis vet hur många element den har måste din utskriftsfunktion alltså gå igenom listan och leta efter slutet. Givet ett första listelement (huvudet) får funktionen skriva ut det elementet och sedan via ett rekursivt anrop skriva ut resten. Som alltid i rekursion ska funktionen ha ett basfall (där den inte rekurserar) och ett fall där den rekurserar, och anropar sig själv med ett mindre problem (mindre i meningen ”närmare basfallet”).

Ett testprogram

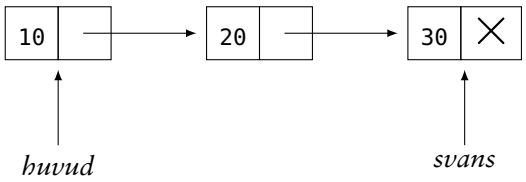
Efter att ha implementerat utskriftsfunktionen ska du skriva ett program som läser in en sekvens med tal från användaren, bygger en lista med dem, och skriver ut listan på skärmen (separerade med mellanrum). Ditt program ska läsa in talen via kommandoraden (alltså via `argc` och `argv`). Använd funktionen `atoi` från stdlib.h för att konvertera från en sträng till ett heltal.

Med `push_front` bör alltså resultatet bli som följer.

```
./task_1 10 20 30
30 20 10
```

```
/*
 * Lägger till ett tal efter ett annat i en int_list.
 *
 * Parametrar:
 *
 * list
 * Pekare på det element där det nya ska läggas in. Det kan vara
 * en tom lista.
 *
 * data
 * Det nya värdet.
 *
 * Returnerar:
 * En pekare på det nya elementet, om det lyckades.
 * NULL annars.
 */
int_list * add_after( int_list * list, int data);
```

Kodlistning 2: Specifikation för `add_after`.



Figur 1: En länkad lista, med huvud och svans.

Som vanligt, dokumentera ditt program (alla funktioner, inklusive `main`) så att det tydligt framgår hur programmet hanterar olika fall. (Vad gör programmet om det får noll argument? Om något argument inte är numeriskt?)

3.2 Bygga listan bakifrån

Skriv nu en funktion som lägger till ett element *efter* ett annat i en lista. Se kodlistning 2. Den här är lite knepigare att implementera än `push_front`. Hur ska du hantera fallet när man försöker lägga till ett element i slutet av en tom lista?

Det är också lite knepigare som användare att bygga listan bakifrån med hjälp av `add_after`. Nu måste du ha dels en pekare som håller reda på var listans *svans* är (alltså det sista elementet), och dels en pekare som håller reda på *huvudet*. Du måste veta var svansen är för att kunna lägga till ett element där, om du inte ska behöva stega igenom hela listan varje gång du ska lägga till ett nytt element; och var huvudet är för att kunna skriva ut hela listan. Se även fig. 1.

Kopiera din funktion för att skriva ut en lista från uppgift 3.1.

Ditt program ska sedan bete sig som i exemplet nedan.

```
./task_2 10 20 30
10 20 30
```

3.3 Modulär länkad lista

Den länkade listan som du nu har implementerat kanske är användbar även i andra delar av spelet. Du kanske också kommer att vilja ha listor av spelar-ID (om det blir ett online-spel där flera spelare kan ansluta eller lämna spelet) eller listor av saker som ligger på en viss plats. Då är det praktiskt att separera listimplementationen från testprogrammet som du skrev i den förra uppgiften, och lägga den i en egen modul, som sedan kan inkluderas av andra. (Om inte annat så kändes det kanske onödigt att behöva kopiera funktionen för utskrift från 3.1 till 3.2!)

I den här uppgiften ska du inte implementera någon ny funktionalitet, utan bara dela upp din tidigare kod i tre filer, samt använda *CMake* för att kompilera och länka de olika delarna.

Dela upp koden

Börja med att flytta alla funktionsdefinitioner som har med listan att göra till en fil `list.c`. Skapa också en fil som heter `list.h` som innehåller funktions-*deklarationerna* från `list.c` men inga funktionskroppar. Både `task_3.c` och `list.c` ska inkludera `list.h` med `#include "list.h"`.

CMake

När ett program ligger uppdelat i olika kompileringsenheter på det här sättet så kan man köra gcc en gång för varje C-fil, och en gång för att länka ihop dem. Det blir snart väldigt opraktiskt, och i verkliga livet ser man till att automatisera det här. Vi ska i fortsättningen använda ett *byggverktyg*, närmare bestämt ett som heter CMake, som bland annat kan kompilera flera filer på samma gång på ett enkelt sätt.[1] Det finns olika byggverktyg i olika miljöer. Poängen med CMake är att det är *cross-platform*, vilket betyder att med samma uppsättning kommandon kan du skapa byggfiler för Linux (GNU make), eller Windows (Visual Studio), eller vad man nu vill använda.

Först behöver du skapa konfigurationsfilen för CMake, där du sedan bestämmer vilka filer som hör till vilket program. Filen ska heta `CMakeLists.txt`. Kom ihåg att det är noga med små och stora bokstäver i filnamn. Nu ska din katalogstruktur se ut så här:

```
inlupp3/build/
inlupp3/CMakeLists.txt
inlupp3/task_1.c
inlupp3/task_2.c
inlupp3/task_3.c
inlupp3/list.c
inlupp3/list.h
```

Innehållet i `CMakeLists.txt` kan se ut så här.

```
# Kommentarer i CMakeLists.txt börjar med brädgård.

# Bör speca vilken version av CMake vi använder, för att slippa varning
cmake_minimum_required(VERSION 3.5)

# Våra flaggor till GCC.
set(CMAKE_C_FLAGS "-std=c99 -Wall -g")

# Program nr 1 heter task_1 och behöver filen task_1.c, och inget annat.
add_executable(task_1 task_1.c)

# Motsvarande för program nr 2
add_executable(task_2 task_2.c)
```

[1] Ännu större poäng blir det att använda byggverktyg när ens program består av flera olika delar som var och en tar lång tid att kompilera. Då ser verktyget till att bara kompilera om de delar som ändrats, eller som är beroende av det som ändrats sedan sist.

```
# Program nr 3 heter task_3 och behöver bygga ihop både task_3.c och list.c.
add_executable(task_3 task_3.c list.c)

# ...

add_executable(task_6 task_6.c)

# Uppgift 3.6 behöver länkas med m, för att kunna använda sqrt.
# (Detta gör samma sak som '-lm'.)
target_link_libraries(task_6 m)
```

När du har den här filen, kan du sedan göra `cd build`, `cmake ..`, `make` för att bygga alla program du har definierat i din `CMakeLists`. Kör du `cmake` och `make` från katalogen `build` så kommer programmen också att hamna där, och inte blandat med dina källkodsfiler. Det kallas *out of source builds*, och är en strategi som man alltid bör använda så fort man arbetar med programmeringsprojekt som består av mer än en enstaka fil! Efter att ha kört `cmake` en gång (för att initiera) kör du alltså bara `make` i fortsättningen, för att bygga dina program.

Ett testprogram

Programmet `task_3` ska göra precis samma sak som `task_2`!

3.4 Stack som abstrakt datatyp

En *stack* är en datastruktur som lagrar en sekvens av data, och som tillhandahåller (åtminstone) två funktioner: `push`, som lägger ett element först (överst) i stacken (stapeln) och `pop` som tar bort det första elementet. (Det är med andra ord en *LIFO*-kö: ”last in first out”.) Man kan använda en stack i många olika sammanhang, men ett exempel är för att göra en miniräknare: ett program som kan beräkna olika sammansatta uttryck och funktioner. I uppgift 3.6 ska du skapa ett sådant program, men vi tar det steg för steg.

I den här uppgiften ska du implementera en stack som innehåller flyttal med typen **`double`**, med hjälp av en länkad lista.

Implementera datastrukturen

Skapa en fil `double_stack.h` och `double_stack_list.c`. H-filen ska innehålla *specifikationen* för stack-datatypen, men ingenting om själva *implementationen*. Implementation ska i stället ligga i C-filen. För att kunna gömma undan implementationen får du i H-filen deklarera följande

```
typedef struct double_stack_struct double_stack;
```

Vi säger alltså att `double_stack` ska vara samma sak som typen **`struct`** `double_stack_struct`. Däremot så är det här en *okomplett datatyp*, eftersom vi bara har *deklarerat den* och inte sagt vad structen faktiskt innehåller! *Men*, så länge vi bara hanterar *pekare* till `double_stack` så spelar det ingen roll. Däremot kommer vi inte att kunna dereferera en sådan pekare förutom i koden som ligger i `double_stack_list.c`. Du får skriva ned den faktiska implementationen av **`struct`** `double_stack_struct` i C-filen:

```
struct double_stack_struct
{
  /**/
};
```

Din **`struct`** kommer att vara väldigt lik den från uppgift 3.3.

En stackfabrik

För att kunna göra datatypen *abstrakt* så behöver du implementera en funktion `create_stack` som skapar en tom stack. Detta för att inte användaren ska behöva veta vilken datastruktur som stacken använder sig av internt. (Eftersom man utanför filen `double_stack_list.c` inte kan använda den okompletta typen, utan bara pekare till den, måste vi på det här sättet ge användaren en pekare som hen kan

använda.) Den här funktionen utgör alltså en sorts ”stackfabrik”, som ger ut en pekare till en ny *tom* stack, som den som vill använda stacken kan jobba med. Låt funktionen ha följande prototyp:

```
double_stack* create_stack();
```

Den här funktionen ska alltså returnera en pekare till en tom lista. (Hur representerar du en tom lista?)

Testa att du kan bygga ett litet testprogram som använder din kod, och bara inkluderar `double_stack.h` samt skapar en ny `double_stack_ptr` med hjälp av `create_stack`.

```
#include "double_stack.h"

int main( int argc, char *argv[] )
{
  double_stack* stack;
  stack = create_stack();
  return 0;
}
```

Det här programmet ska gå att kompilera och länka, även om det inte ger något synligt resultat när det körs.

Push

Implementera funktionen `push` för att lägga nya element på stacken.

```
double_stack* push( double_stack* stack, double d );
```

Den här funktionen tar alltså en pekare till en (eventuellt tom) stack, samt ett tal, som input, och returnerar en pekare till toppen på stacken efter att talet `d` har lagts till. Den kommer att vara väldigt lik `push_front` från uppgift 3.1.

Nu kan ditt testprogram se ut så här.

```
#include "double_stack.h"

int main( int argc, char *argv[] )
{
  double_stack* stack;
  stack = create_stack();   // tom stack
  stack = push( stack, 1 ); // nu innehåller stacken 1
  stack = push( stack, 2 ); // nu innehåller stacken 2, 1
  return 0;
}
```

Pop

Implementera också funktionen `pop`

```
double_stack* pop( double_stack* stack, double * d);
```

som tar bort det översta elementet i stacken och lägger det där `d` pekar.

Ett testprogram

Nu har du i din `double_stack.h` ett fullständigt gränssnitt för en stack som innehåller flyttal, med alla de funktioner som behövs. Skapa nu slutligen ett testprogram som lägger in några tal (ta dem från `stdin`) i en stack, och sedan `pop`-ar ett och ett och skriver ut dem, vart och ett på en ny rad.

```
./task_4 1 2 3.4 5
5.000000
3.400000
2.000000
1.000000
```

Eftersom det inte finns någon funktion som kan kolla om stacken är tom eller ej måste du själv hålla reda på att du `pop`-ar lika många element som du har `push`-at.

Glöm inte att dokumentera vilka tester du har gjort för att verifiera att stacken fungerar korrekt.

3.5 En annan stackimplementation

Nu får vi tänka oss att en prestandaanalys av ett större program som använder din stackimplementation har kommit fram till att det tar för mycket tid att pusha och poppa en stor mängd element, och ett förslag på optimering är att i stället för en länkad lista ha ett *fält* som underliggande datatyp för stacken.

Gör om implementationen av din stack. Tack vare att det är en abstrakt datatyp där implementationen är åtskild kan du ändra utan att förstöra för existerande kod som använder din stack-implementation!

Det du behöver göra är att skapa en ny fil `double_stack_array.c` som, precis som `double_stack_list.c` inkluderar `double_stack.h`, men implementerar de tre funktionerna med hjälp av fält i stället för en länkad lista.

I det här fallet kommer din **`struct`** `double_stack_struct` att behöva innehålla ett **`double`**-fält med "lagom många" element i stället för en pekare till en lista. Dessutom behöver **`struct`**-en innehålla en räknare som visar vilket index i fältet som är toppen på stacken för tillfället.

Om du vill kan du använda ett fält som är dynamiskt allokerat (med `malloc`) i stället. Om du gör på det sättet kan du också med hjälp av funktionen `realloc`[2] göra fältet större vid behov.

Kopiera din `task_4.c` till `task_5.c` och verifiera att du får samma resultat! (Tänk på att länka med rätt fil i `CMakeLists.txt`: alltså `double_stack_array.c` i stället för `double_stack_list.c`.)

Fråga 1
- Vad gör din implementation när det blir stack overflow? (Alltså när någon försöker pusha ett element när hela fältet är fullt.) Testa gärna och se!

Fråga 2
- Hur skiljer sig det från versionen med en länkad lista i stället för ett fält? Beter de sig likadant eller inte?

Fråga 3
- Vad gör din implementation när det blir stack *underflow*? (Alltså när du poppar från en tom stack.) Testa gärna och se!

Det är inte nödvändigt att ha felkontroll så att programmet "gör det rätta" i fallen som frågorna ovan handlar om, men du behöver fundera på vad som kan händer i de olika fallen.

Frivilligt

3.6 Miniräknare

För att få chansen att också *använda* din stack-implementation ska du nu implementera en miniräknare som använder omvänd polsk notation.[3] En sådan miniräknare tar ett argument i taget och lägger på en stack, och när en operator (plus, minus, etc.) ges som argument så appliceras den på de två översta stackelementen.

Din miniräknare ska hantera de fyra räknesätten (addition, subtraktion, division, multiplikation) och använda **`double`** för beräkningarna.

Ett exempel på en instruktionssekvens är så här:

1. läs input från *standard input*,
2. om det är numeriskt (använd någon av funktionerna från stdlib.h[4] för att kolla det), pusha det på stacken,
3. annars (det är en operator, t ex plus eller minus):

[2] Se t ex https://www.tutorialspoint.com/c_standard_library/c_function_realloc.htm
[3] Se också https://sv.wikipedia.org/wiki/Omvänd_polsk_notation.
[4] Se t ex http://www.fortran-2000.com/ArnaudRecipes/Cstd/2.13.html

(a) poppa två element från stacken,

(b) applicera operatorn på de båda,

(c) pusha tillbaka resultatet på stacken.

Användaren kan avsluta programmet med *end of file* som man skickar genom att trycka ctrl-d på tangentbordet. Då kommer `scanf` att returnera konstanten `EOF`. Implementera alltså en loop som kör så länge som `scanf` inte returnerar `EOF`.

Nedan följer ett exempel på vad din miniräknare ska kunna göra. Allt som står är inmatning från användaren (i kommandoprompten), utom raderna med `=`, som visar utmatning från programmet. Se till att ditt program följer samma struktur.

```
./task_6
4
3
/
= 1.333333
10.6
-
= -9.266667
```

Om felhantering

Du måste hantera om en operator läggs på en stack med mindre än två element (stack underflow), så att en användare kan göra lite fel och ändå köra vidare utan att programmet kraschar. Till exempel ska `3 + 4 +` ge resultatet 7 – trots att det första plustecknet är en felaktig inmatning, eftersom det då bara ligger ett element på stacken. Det första plustecknet är fel, så stacken återställs. När det andra plustecknet kommer ligger 3 och 4 på stacken, och resultatet blir 7. Hur du ska kolla om en `pop` lyckas eller ej beror på hur du implementerat din stack i de tidigare uppgifterna.

Däremot behöver du inte ta hänsyn till en användare som medvetet försöker mata in konstiga saker till programmet, i den här uppgiften. Du kan förutsätta att det bara kommer numeriska värden eller något av `+ - * /`.